Performance Trade-offs Implementing Refactoring Support for Objective-C

Robert Bowdidge*
rbowdidge@mac.com

Abstract

When we started implementing a refactoring tool for real-world C programs, we recognized that preprocessing and parsing in straightforward and accurate ways would result in unacceptably slow analysis times and an overly-complicated parsing system. Instead, we traded some accuracy so we could parse, analyze, and change large, real programs while still making the refactoring experience feel interactive and fast. Our tradeoffs fell into three categories: using different levels of accuracy in different parts of the analysis, recognizing that collected wisdom about C programs didn't hold for Objective-C programs, and finding ways to exploit delays in typical interaction with the tool.

Categories and Subject Descriptors D.2.6 [Software Engineering]: Programming Environments

General Terms Design, Language

Keywords: refactoring, case study, scalability, Objective-C

1. Introduction

Taking software engineering tools from research to development requires addressing the practical details of software development: huge amounts of source code, the nuances of real languages, and multiple build configurations. Making tools useful for real programmers requires either addressing all these sorts of issues, or accepting various trade-offs in order to ship a reasonable software tool.

In our case, we wanted to add refactoring to Apple's Xcode IDE (integrated development environment.) [1] The refactoring feature would manipulate programs written in Objective-C. Objective-C is an object-oriented extension to C, and Apple's primary development language [1]. In past research [2], I'd found it acceptable to take multiple minutes to perform a transformation on a small Scheme program. The critical requirements for our commercial tool were quite different:

- **Support the most common and useful transformations.** Renaming declarations, replacing a block of code with a call to a new function, and moving declarations up and down a class hierarchy were mandatory features.
- **Refactor 200,000 line programs**. The feature had to work on real, medium-sized applications. The actual amount of code to parse was much larger than the program's size. Most Mac OS X compilation units pull in headers for common system libraries, requiring at least another 60-120,000 lines of code that would need to be parsed for every compilation unit. Such large sets of headers are not unique to Mac OS X. C programs using large libraries like the Qt user interface library would encounter similar scalability issues.
- **Interactive behavior**. Xcode's refactoring feature would be part of the source code editor. Users will expect transformations to complete in seconds rather than minutes, and the whole experience would need to feel interactive [3]. Parsing and analyzing programs of this size in straightforward ways would result in an unacceptable user experience. In one of my first experiences with a similar product, renaming a declaration in a 4,200 line C program (with the previously-mentioned 60,000 lines of headers) took two minutes.
- **Don't force the user to change the program in order to refactor**. The competing product previously mentioned could provide much more acceptable performance if the user specified a pre-compiled header—a single header included by all compilation units. However, converting a large existing project to use a pre-compiled header is not a trivial task, and the additional and hidden setup step discourages new users.
- **Be aware of use of C's preprocessor**. The programs being manipulated would make common use of preprocessor macros and conditionally compiled code. If we did not fully address how the preprocessor affected refactoring, we would at least need to be aware of the potential issues.
- **Reuse existing parsing infrastructure.** We realized there wasn't sufficient time or resources to write a new parser from scratch. Analysis would need to be done by an existing Objective C parser used for indexing global declarations.

Refactoring had to work best for our third-party developers—primarily developers writing GUI applications. It should also work well for developers within Apple, but not for those writing low-level operating system or device driver code.

Performance and interactivity were key—we wanted to provide an excellent refactoring experience. In order to meet these performance and interactivity goals, we attacked three areas: using different levels of accuracy in different parts of the tool, recognizing differences between our target programmers and typical C programmers, and finding ways to exploit delays in the user's interaction with the tool.

2. Different Levels of Accuracy

In C, each source file is preprocessed and compiled independently as a "compilation unit". Each can include different headers, or can include the same headers with different inclusion order or

* This work was performed while the author was at Apple, and discusses the initial implementation of refactoring for Xcode 3.0. The author is currently at Google.

3rd Workshop on Refactoring Tools '09, Oct. 26, 2009, Orlando, FL.

initial macro settings. As a result, each compilation unit may interpret the same headers different ways, and may parse different declarations in those same headers. For correct parsing, the compiler needs to compile every source file independently, read in header files anew each time, and fully parse all headers.

For small programs, this may not matter, but with Mac OS X, each source file includes between 60-120,000 lines of code from header files. Precompiled headers and other optimizations could speed compile times, but not all developers use precompiled headers, nor could we demand that developers use such schemes in order to use refactoring. Naively parsing all source code was not acceptable; we saw parse times of around five seconds to parse a typical set of headers, so five seconds minimum per file per build configuration would be completely unacceptable.

We realized two facts about programs that made us question whether we needed compilation-unit-level accuracy. We realized that although programmers have the opportunity for header files to be interpreted differently in each compilation unit, most programmers intend for the headers to be processed the same in all compilation units. (When header files are not processed uniformly, it can cause subtle, nasty bugs that can take days to track down.) We also realized that system header files are not really part of the project, and not targets for refactoring. We needed to correctly parse system header files merely for their information on types and external function declarations. For most refactoring operations, we didn't care if the `my_integer_t` type was 4 bytes long or 8; we just needed to know that the name referred to a type. We also knew that correct refactoring transformations shouldn't change the write-protected system header files.

We thus made two assumptions about headers we parsed. First, we decided to parse each header file at most once, and would assume that the files were interpreted the same in each compilation unit. This meant that we could shorten parsing times for at least five seconds per file to five seconds (for all system header files), plus the additional time to only parse the source files and headers in the project.

Second, we gathered less position information for system header files. We knew that changes in system header files were both incorrect (because we couldn't change the existing code in libraries) and uninteresting (because we couldn't change all other clients of the header file.) We gathered less exact position information for such files, and would flag errors if a transformation would change code in a system header file.

We also realized that the user interface needed information about the source code to identify whether refactoring was possible for a given selection, which transformations were possible, and what the default parameters for the transformation would be. Because we wanted the user interface to make these suggestions immediately without waiting for parsing to complete, we used saved information from the Xcode's declaration index when helping the user propose a refactoring transformation. We did have some issues where indexer information had inaccuracies (when its less accurate parser misparsed certain constructs), but in general we found the information good enough for our first release.

3. The Typical Programmer

Dealing with conditional code and multiple build configurations is another major issue for refactoring and source code analysis of C programs. We realized that many of the assumptions about C code did not hold for Objective-C programs, and changed our expectations of what we would implement.

C's preprocessor supports conditional code—code only compiled if certain macros are set. Although some conventions exist for using conditional directives, the criteria triggering a particular block of code usually can be understood only by evaluating the values of the controlling macros at the point the preprocessor would have interpreted the directive. If source code with conditional code was refactored without considering all potential conditions, syntax errors or changed behavior could be introduced.

Others have proposed various solutions for handling conditional code. Garrido and Johnson expanded the conditional code to cover entire declarations, and annotated the ASTs to mark the configurations including each declaration [4]. Vittek suggested parsing only the feasible sets of configuration macros, parsing each condition separately, and merging the resulting parse trees [5]. McCloskey and Brewer proposed a new preprocessor amenable to analysis and change, with tools to migrate existing programs to the new preprocessor [6].

We instead chose to parse for a single build configuration—a single set of macros, compiler flags, and include paths. Parsing a single build configuration appeared reasonable because Objective-C programs use the preprocessor less than typical C programs, because occurrences of conditional code were unlikely to be refactored, and because remaining uses of conditional code were insensitive to the refactoring changes.

Ernst's survey of preprocessor use found that UNIX utilities varied in their use of preprocessor directives. He found the percentage of preprocessor directives to total non-comment, non-blank (NCNB) lines ranged between 4% and 22% [7]. By contrast, only 3-8% of lines in typical Objective-C programs were preprocessor directives. (Measurements were made on sources for the Osirix medical visualization application, Adium multi-protocol chat client, and Xcode itself.)

Within those Objective-C programs, preprocessor directives and conditional code also occurred much more frequently in the code unlikely to be refactored. Many were either in third-party utility code, or in cross-platform C++ code. The utility code was often public-domain source code intended for multiple operating systems. Such code is unlikely to be refactored for fear of complicating merges of newer versions. For applications designed for multiple operating systems, often a core C++ library would be the basis of all versions, and separate user interface code would be written for each operating system. Because our first release would not refactor or parse C++ code, such core code would be irrelevant to refactoring. For the Objective-C portions of the projects, only 2-4% of all lines were preprocessor directives.

The preprocessor directives that do appear in Objective-C code are often irrelevant to refactoring. Of Ernst's eleven categories of conditional code, many are either unlikely to affect the target audience, or are irrelevant to refactoring in general. Include guards are less frequently used in Objective-C because a separate directive (#import) ensures a file is included only once. Conditional directives that always disabled code ("#if (0)") can be handled in the same way comments are processed. Operating system -specific conditional code is unlikely in Objective-C code because the language is used only on Mac OS X.

However, there are three problematic conditional code directives that appear in Objective-C programs: code for debugging, architecture-specific code, and conditional code for enabling and disabling features in the project.

Conditional code for debugging is unlikely to be troublesome. The rename transformation will make an incorrect change if a declaration is referenced in conditional code that is not parsed. If the condition is parsed, then the conditional code is not a concern. The most dangerous case occurs when code that needs to be manipulated exists in two conditionally compiled sections of code never parsed at the same time. Luckily, most conditional code controlled by debugging macros only adds code to the debug case,

and does not add code to the non-debug case. As long as we parse the program with debug macros set (which should be the default during development), then we should parse all necessary code.

Architecture-specific code is more common at Apple because we support two architectures (x86 and PowerPC), both in 32 and 64 bit versions. Most of the architecture-specific conditional code is found in low level system code and device drivers. The external developers we are targeting with refactoring would be working on application software, and would be unlikely to have architecture-specific code.

Project-specific features controlled by conditional compilation directives represent a larger risk. Some of these may actually be in use (such as code shared between an iPhone and Mac application), and others may represent dead code. Code may exist on both sides of a condition. For the first release, we only changed code in the current build configuration, and relied on the user to be aware of and avoid changes in project-specific conditional code.

4. Exploiting Interaction Delays

A final area for optimization was deciding when parsing and refactoring work would begin during actual use. Even with our previous decisions, parsing speed still wasn't acceptable. Our rough numbers were that we could parse all the system header files in about 5 seconds, and then could parse an additional ten files a second on a typical machine. Caching the results of the header file parsing was an obvious solution, but we weren't sure we had the time to implement such caching.

A straightforward implementation would start parsing after the user specified the transformation to be performed, and only show results when the transformation was complete. We realized we could speed perceived performance by starting parsing early, and showing partial results before the transformation completed.

4.1 Optimistically Starting Parsing

It usually takes a few seconds for a programmer to specify a refactoring transformation. Even for the simple rename, the user needs to indicate that he wants to rename a declaration, then needs to type in the new name. For "extract function", the additional choices for parameter name and order requires additional time.

To improve perceived performance, we began parsing the currently active file and header files as soon as the programmer had selected the "refactor" menu item. For refactoring transformations that only affected a single file, this often meant that as soon as the user specified the parameters for refactoring, the parsing had already been completed, and the transformation would be ready immediately.

4.2 Showing Partial Results

When performing transformations changing multiple files, we similarly exploited how programmers would interact with the refactoring tool. We knew that most programmers beginning to use refactoring might want to examine the changes being made to double-check that the transformation was correct. If we assumed that most transformations would be successful (because the programmer was unlikely to try a transformation they thought would break their code), then we could begin showing partial results immediately rather than waiting for the entire transformation to be complete and validated to be safe.

Most descriptions of refactoring break each transformation into two parts: the pre-conditions (which indicate the requirements that must be met before a transformation may be performed) and the changes to the source code (which are only performed after the change is believed safe. [8]) Because parse times are liable to be longer than a few seconds, the "check, then perform" approach would not have been interactive. The user would have to wait until all source code was parsed and all refactoring complete before examining any results. Similarly, parse trees for all functions would need to be generated before any refactoring work could begin. If the project being manipulated was particularly large, then the parse trees could consume huge amounts of memory.

To make refactoring more palatable on large projects, we designed our transformations to work in several phases so that changes could be presented shown after only some of the code had been parsed and portions of the transformation performed. (See Figure 1). We also could dispose of some parse trees as soon as that code has been analyzed. The seven phases for our transformations are:

- **check user input**: precondition checks that could be done with the initial inputs to the transformation only.
- **check first file**: precondition checks to do after the file containing the selection is parsed. Generally, the analysis performed in this phase only performs initial sanity checks requiring parse trees. For the rename transformation, the phase checks that the declaration can be renamed, if the name is a valid C identifier, and if the declaration is not in a system header file.
- **perform first file**: apply any changes that can be determined after the first file is parsed. Few transformations do work in this phase.
- **check per-file**: precondition checks to do after parsing each compilation unit.
- **perform per-file**: changes to apply after parsing each compilation unit. Most transformations do the bulk of their work in the per-file category. The check and perform parts both look at newly found uses of relevant declarations, and make appropriate changes. Each transformation specifies if the memory for parsed representations of function bodies can be freed before beginning the next file.
- **check final**: precondition checks to do after parsing all files. The after-parsing checks tend to involve existence tests or non-existence tests—whether any situations exist that indicate the transformation is unsafe such as "did we ever see any declara-

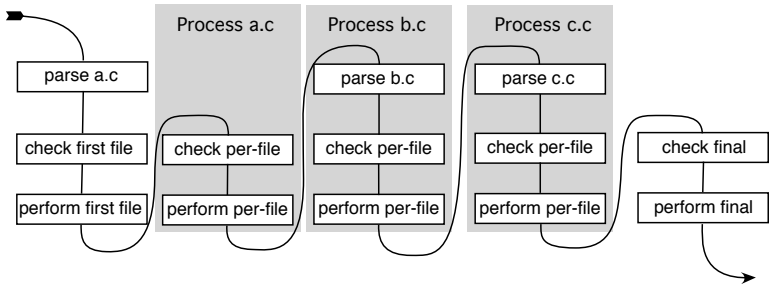

Figure 1: Order of processing of interleaved refactoring transformation on three source files a.c, b.c, and c.c. Results of the transformation are incrementally updated after each perform- phase is complete.

tions with this name already?" Some of these checks could be done incrementally as each file is parsed.

- **perform final**: changes to apply after parsing all files. The perform final phase is typically used for edits that cannot be constructed until all sources have been parsed. For example, when converting references to a structure's field to call getter or setter functions, the transformation needs to determine where to place the new accessor functions. The accessors need to be placed in a source file (rather than a header), preferably near existing references to the field or the definition of the structure. Typically, the transformation can place the functions as soon as a likely location is found. If no appropriate location for the new code is found in any source file, the perform final phase chooses an arbitrary location.

By breaking up each transformation in this way, the user experience of refactoring becomes more interactive. The refactoring user interface can show the list of files which must be parsed for a transformation. As each file is parsed and changes are identified, the user interface indicates completion and notes the number of changes in that file. Selecting the filename shows a side-by-side view of the source before and after the change. As the transformation progresses, more files and edits are displayed. The user can examine proposed changes as soon as each file is processed. While examining the changes, the user can also choose not to include some changes, or can make additional edits to the changed source code. In this way, the user can both measure progress and can be working productively as the transformation progresses.

The interleaved transformation approach has the risk of declaring a transformation unsafe after the user has already examined some changes. This turned out not to be a problem in actual use. Programmers weren't bothered by the delayed negative answer. We also found very few transformations where we could outright refuse to do a transformation. We might warn the result is incorrect, but we found programmers often wanted the chance to apply those incorrect changes and then fix remaining problems with straight edits.

5. Conclusions

Overall, our progress on refactoring matched effort described on similar projects. Our first prototype was completed in three months by one person, and our first release required two years and three people. We found the transformations tended to be easy to write. Most of our parsing effort focused on scalability - getting parsing performance and memory use low, and making sure it worked well inside the IDE. We also found that implementing a polished user interface took the majority of the overall effort, with two of the engineers working full time on refactoring workflow and on making the file comparison view as polished as possible.

With the trade-offs described here, we met our performance goals. Our goal at the beginning of the project was to permit refactoring on 200,000 line projects, and be able to rename a frequently-referenced declaration within 30 seconds. On a 2.2 GHz Dual Xeon PowerMac with 1 GB of memory, we renamed declarations in a 270,000 line Objective-C project. We found we could rename a class referenced in 382 places through 123 files in 28 seconds. We could rename a class used in 65 files in 15 seconds. Operations involving only a single file took around 8 seconds; this time was irrespective of the source file because parsing the headers dominated. Most transformations only require parsing a small subset of source files in a project. However, one of the transformations searches all code for iterators that can be converted to use a new language feature. Parsing the entire 270,000 line project for this transformation takes around 90 seconds. This is not acceptable for the interactive transformations, but is adequate for an infrequently run transformation that changes all source files. The refactoring feature as described shipped as part of Xcode 3.0 and Mac OS X 10.5.

Building software development tools in industry requires making tradeoffs in both requirements and design. Some are driven by the expected needs of users such as the size of programs to be refactored, or response times expected. Some are driven by scalability issues such as whether to save pre-processed header files in the IDE between refactorings, or whether to re-parse headers from scratch each time. Other tradeoffs occur for business, timing, or staffing reasons, affecting whether a feature might even be implemented, or whether a new parser is written from scratch.

As described in this paper, our requirements strongly affected what we could and did implement. The particular tradeoffs we made may not appear to be the "right" or "perfect" decision in all cases, but they are representative of the sorts of decisions that must be made during the process of commercial development. Our three themes of trade-offs—identifying where different levels of accuracy were acceptable, recognizing differences between "our typical user" and "a typical user", and exploiting delays in user interaction to improve responsiveness—suggest ways that other tools can meet their own goals.

Acknowledgements

Thanks to Michael Van De Vanter and Todd Fernandez for their feedback on a previous version of this paper. Dave Payne originally suggested applying the transformations file-by-file. Andrew Pontious and Yuji Akimoto implemented the refactoring user interface, and kept us focused on an interactive experience.

Our approach for incrementally showing refactoring results is also described in U.S. Patent Application 20080052684, " Stepwise source code refactoring".

References

[1] Apple, "Apple Developer Documentation: Objective-C Programming Language," Cupertino, CA 2007.

[2] R. W. Bowdidge and W. G. Griswold, "Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization," ACM Transactions on Software Engineering and Methodology, vol. 7(2), 1998.

[3] D. Bäumer, E. Gamma, and A. Kiezun, "Integrating refactoring support into a Java development tool," in OOPSLA 2001 Companion, 2001.

[4] A. Garrido and R. Johnson, "Analyzing Multiple Configurations of a C Program," in 21st IEEE International Conference on Software Maintenance (ICSM), 2005.

[5] M. Vittek, "Refactoring Browser with Preprocessor," in 7th European Conference on Software Maintenance and Reengineering, Benevento, Italy, 2003.

[6] B. McCloskey and E. Brewer, "ASTEC: a new approach to refactoring C," in 13th ACM SIGSOFT international symposium on Foundations of Software Engineering ESEC/FSE-13, 2005.

[7] M. D. Ernst, G. J. Badros, and D. Notkin, "An Empirical Analysis of C Preprocessor Use," IEEE Transactions on Software Engineering, vol. 28, pp. 1146-1170, December 2002.

[8] W. F. Opdyke, "Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks," University of Illinois, Urbana-Champaign, 1991.